

Materialien zu “Einführung ins Programmieren”

Armin Scrinzi
(Work in progress)

October 22, 2024

Dies ist kein Skript, sondern eine Sammlung von Lernmaterialien und Tabellen samt einigen Anmerkungen. Die Vorlesung findet heuer erstmals statt und der genaue Verlauf wird aus der Praxis bestimmt. Für vollständige Inhalte ist die Teilnahme an der Vorlesung notwendig.

Contents

1	Sprache, Notation und Konventionen	4
1.1	Online Tutorials	4
2	Schnellstart	4
2.0.1	Linuxsystem EinProg.oiva herunterladen und starten	4
2.0.2	“helloStudent” — ein Programm schreiben und ausführen	4
3	Grundlagen	4
3.0.1	g++ — Compile und Link	4
3.0.2	Directories und Files	4
3.0.3	Directory tree — Verzeichnisbaum	5
3.0.4	Mouse und Pointer in Linux	5
3.0.5	Command Line Interface — arbeiten mit dem Terminal	6
3.0.6	Filesystem	7
3.1	multiply.cpp — Eingabe, Multiplizieren, Ausgabe	7
3.2	Tests	8
4	IDE — Integrated Development Environment	9
5	Operatoren	10
6	Bedingte Ausführung: if und else	11
7	for-Loops	12
8	Variablentypen: int, long, float, double, char, std::string	13
8.1	Ganze Zahlen: int und long	13
8.2	Einzelne Bytes = 8 Bit: char	13
8.3	Zeichenketten: std::string aus der Standard Library	15
8.4	Floating point (=Fließkomma) Variable: float und double	15
8.5	Truncation errors	15
9	Header Files	16
10	Phasen des Programmierens und ihr Zeitaufwand	17
11	Hardware	17
12	Programmierstil: “good programming practice”	18
12.1	Variablen namen	18
12.2	Headerfiles und #include	19

13 Advanced topics (example Schrödinger equation)	20
13.1 <code>git</code> — maintaining complex codes	20
13.2 Classes	20
13.3 Inheritance	20
13.4 Working with pointers	20
13.5 Templates	20
13.6 Containers: <code>std::set</code> , <code>std::map</code> , etc.	20
13.7 Parallelization: OpenMP and MPI	20

List of Tables

1	Englische Begriffe und deutsche Entsprechungen	4
2	Konventionen und Notation	4
3	Auswahl einiger Befehle im Terminal (Command Line Interface)	6
4	Arithmetische Operatoren	10
5	Zuweisungsoperatoren	11
6	Vergleiche	11
7	Wichtige Variablentypen	13
8	Zeitaufwand beim Programmieren	17

List of Figures

1	Im Filesystem bewegen und Programme ausführen	7
2	Directories und Files erzeugen und entfernen	7
3	<code>multiply.cpp</code>	8
4	IDE am Beispiel <code>multiply.cpp</code>	10
5	Loops: <code>forLoop.cpp</code>	12
6	Fehler: “overflow” von <code>int</code>	13
7	(**)General C++ type structure	14
8	Hardwarestruktur	18

1 Sprache, Notation und Konventionen

Es werden zumeist die englischen Begriffe verwendet, da dies langfristig der Praxis entspricht. Es kommt damit unvermeidlich zu einem unschönen Sprachmix.

Table 1: Englische Begriffe und deutsche Entsprechungen

File	Datei
Directory	Verzeichnis
Executable	Ausführbar(es File)
Prompt	Eingabeaufforderung
Float	Fliesskomma

Table 2: Konventionen und Notation

<code>code</code>	Programmcode oder fester Befehl
<i>some-name</i>	vom User zu spezifizierender Name
> oder \$	“Prompt”, d.h. aktuelle Eingabezeile im Terminal

1.1 Online Tutorials

<https://www.w3schools.com/cpp>

2 Schnellstart

2.0.1 Linuxsystem EinProg.ova herunterladen und starten

2.0.2 “helloStudent” — ein Programm schreiben und ausführen

3 Grundlagen

3.0.1 g++ — Compile und Link

3.0.2 Directories und Files

Directory (oder “folder”), deutsch “Verzeichnis”: Enthält eine Liste von Name, die selbst wieder Directories bezeichnen oder auch Files.

Files:

ASCII (=Text)-Files: sind für Menschen lesbar, weniger Informationsdichte, mehr Speicherplatz, für Maschinenverarbeitung langsam, enthält Buchstaben, Zahlen und Sonderzeichen. Beispiel: `helloWorld.cpp`, der C++ Code vor Kompilation. Den Inhalt mit `$ more helloWorld.cpp` ansehen (oder im Texteditor).

Binary Files: binär, enthalten 0/1 Sequenzen, hohe Informationsdichte, schnell lesbar, aber nur für Maschinen (d.h. andere Programme) sinnvoll. Beispiel: `helloWorld`, das kompilierte Programm

3.0.3 Directory tree — Verzeichnisbaum

Baumstruktur von Directories, da jedes Directory wieder Directories enthalten kann.

3.0.4 Mouse und Pointer in Linux

Verwendung sehr wie in Windows, wichtiger unterschied: *Einfach*click (nicht Doppel!) zum Starten von Programm. Doppelclick startet meistens zwei Kopien des Programms. Rechte Maustaste: zumeist Menü im für gegebenes Fenster Mittlere Maustaste: zumeist “Paste” (Einfügen)

3.0.5 Command Line Interface — arbeiten mit dem Terminal

Einführung in <https://ubuntu.com/tutorials/command-line-for-beginners>

Table 3: Auswahl einiger Befehle im Terminal (Command Line Interface)

Befehl	Verbalisierung	Funktion
<code>ls</code>	“list”	zeige Files im aktuellen Directory
<code>pwd</code>	“present work dir.”	zeige den Namen des aktuellen Directory
<code>./prog</code>	führe im aktuellen Directory befindliches Programm <i>prog</i> aus	
<code>cd name</code>	“change directory”	wechsle in ins Subdirectory <i>name</i>
<code>cd ..</code>		wechsle ins nächsthöhere Directory
<code>vi</code>	“visual”	allgemeiner Linux Editor (auf jedem Linux)
<code>mkdir name</code>	“make directory”	Erzeuge neues Directory <i>name</i>
<code>rm name</code>	“remove”	File <i>name</i> entfernen
<code>rm -r name</code>	<code>rm</code> “recursive”	Directory <i>name</i> entfernen
Ctrl c	“control c”	bricht Programm ab
Ctrl d	“control d”	bricht Programm ab (alternative)
Ctrl z	“control z”	hält Programm an, ohne abzuberechnen
<code>bg</code>	“background”	führe angehaltenes Programm im Hintergrund aus
<code>fg</code>	“foreground”	holt Programm in den Vordergrund
Ctrl r <i>text</i>	“recall”	suche vorhergehende Command Line, die <i>text</i> enthält
<i>partOfName</i> TAB		ergänze, oder zeige mögliche Ergänzungen für <i>partOfName</i>
<code>more file</code>		zeigt Fileinhalt am Terminal (nur ASCII)
Pfeil left/right		bewege Cursor nach links/rechts in Command Line
Pfeil up/down		gehe zur vorherigen/folgenden Command Line
<code>find . -iname "*text"</code>	ausgehend vom aktuellen Directory, suche filenames der <i>text</i> enthält	

3.0.6 Filesystem

```
student@EinProg:~$ pwd ← Befehl: zeige "present working directory"
/home/student ← Resultat: "home directory" für user "student"
student@EinProg:~$ ls ← Befehl: liste Files und Directories im aktuellen Directory
Desktop Documents Downloads HelloWorld Libraries snap
student@EinProg:~$ cd HelloWorld ← wechsele ins Directory "HelloWorld"
student@EinProg:~/HelloWorld$ ls ← ausführbares Programm (compiliert & linked)
helloWorld helloWorld.cpp ← C++ Code
student@EinProg:~/HelloWorld$ ./helloWorld ← Führe "helloWorld" aus
hello world ← Output von "helloWorld" (nicht sehr spannend)
student@EinProg:~/HelloWorld$ pwd
/home/student/HelloWorld ← Wir befinden uns in diesem Directory
student@EinProg:~/HelloWorld$ cd
student@EinProg:~$ pwd
/home/student ← Wir haben ins "home directory" gewechselt
student@EinProg:~$ ls -l HelloWorld/helloWorld.cpp
-rw-rw-r-- 1 student student 95 Sep 12 20:18 HelloWorld/helloWorld.cpp
student@EinProg:~$ ls HelloWorld/helloWorld.cpp ← lange ("-l") Info über helloWorld.cpp im Directory HelloWorld
```

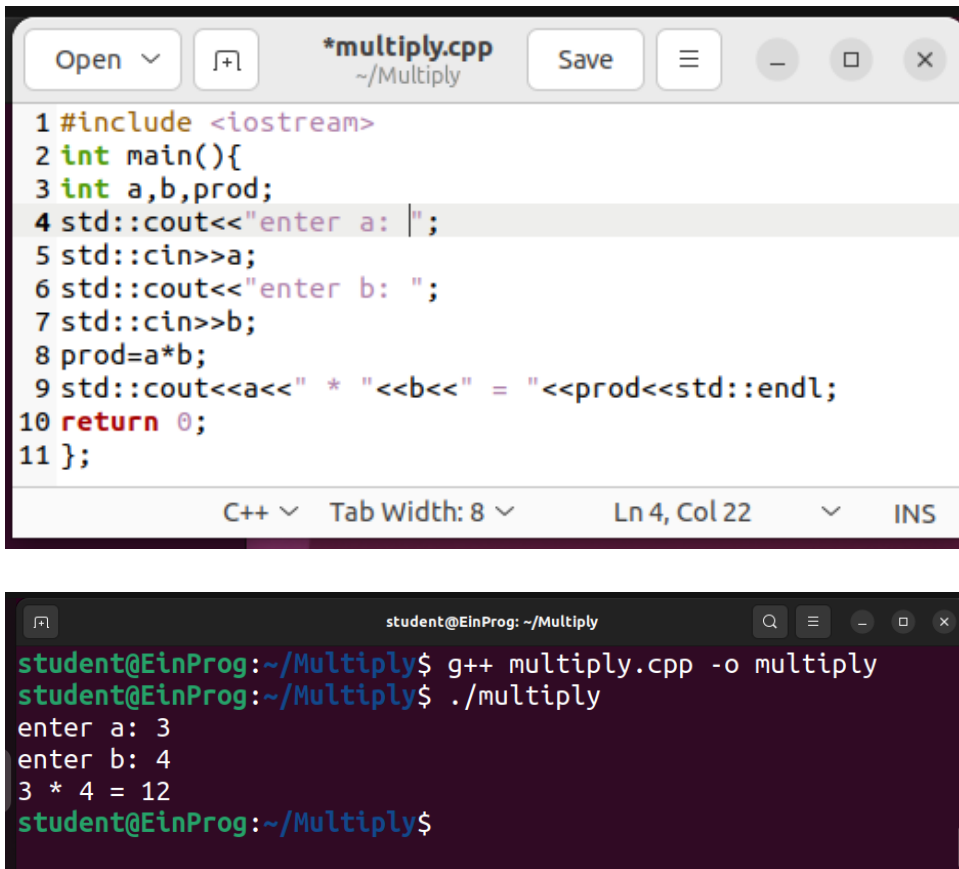
Figure 1: Im Filesystem bewegen und Programme ausführen

```
student@EinProg:~$ ls
Desktop Documents Downloads HelloWorld Libraries snap
student@EinProg:~$ mkdir dumDir ← erzeuge Directory "dumDir"
student@EinProg:~$ ls
Desktop Documents Downloads dumDir HelloWorld Libraries snap
student@EinProg:~$ cd dumDir ← wechsele nach "dumDir"
student@EinProg:~/dumDir$ mkdir dummerDir
student@EinProg:~/dumDir$ ls
dummerDir ← erzeuge dummerDir in dumDir
student@EinProg:~/dumDir$ cd dummerDir
student@EinProg:~/dumDir/dummerDir$ pwd
/home/student/dumDir/dummerDir
student@EinProg:~/dumDir/dummerDir$ ls
Erzeuge leeres File "dumFile"
student@EinProg:~/dumDir/dummerDir$ touch dumFile
student@EinProg:~/dumDir/dummerDir$ ls
dumFile ← wechsele in nächsthöhere Directory
student@EinProg:~/dumDir/dummerDir$ cd ..
student@EinProg:~/dumDir$ ls
dummerDir ← wechsele in nächsthöhere Directory
student@EinProg:~/dumDir$ cd ..
student@EinProg:~$ ls
Desktop Documents Downloads dumDir HelloWorld Libraries snap
student@EinProg:~$ rm -r dumDir ← entferne dumDir und alles darin
student@EinProg:~$ ls
Desktop Documents Downloads HelloWorld Libraries snap
student@EinProg:~$
```

Figure 2: Directories und Files erzeugen und entfernen

3.1 multiply.cpp — Eingabe, Multiplizieren, Ausgabe

- `#include <iostream>` — "header File" der "C++ standard library", hier für in- und output
- Input- und output "streams"
- `int` — ganzzahlige Variable und algebraische Operationen.



The top screenshot shows a code editor window titled `*multiply.cpp` with the following C++ code:

```
1 #include <iostream>
2 int main(){
3     int a,b,prod;
4     std::cout<<"enter a: ";
5     std::cin>>a;
6     std::cout<<"enter b: ";
7     std::cin>>b;
8     prod=a*b;
9     std::cout<<a<<" * "<<b<<" = "<<prod<<std::endl;
10    return 0;
11 };
```

The bottom screenshot shows a terminal window with the following commands and output:

```
student@EinProg: ~/Multiply
student@EinProg:~/Multiply$ g++ multiply.cpp -o multiply
student@EinProg:~/Multiply$ ./multiply
enter a: 3
enter b: 4
3 * 4 = 12
student@EinProg:~/Multiply$
```

Figure 3: multiply.cpp

- `int main()` — Hauptprogramm
- `{ ... code ... }` — Code-Block
- `...code...;` — Semikolon (Strichpunkt) Ende einer logischen Programmzeile

3.2 Tests

1. Erzeugen Sie mittels der Befehle `mkdir` die folgende Directorystruktur, ausgehend vom Home-Directory. Im jeweiligen Directory erzeugen Sie Files wie angegeben mittels des Befehls `touch`
MyTopDir MySubDir
 AnotherSub:file1,file2 SubSubDir:file3 und entfernen Sie an-
OtherTop ToBeRemoved:file1
schliessend OtherTop wieder vollständig.
2. Modifizieren Sie den Code "helloStudent.cpp" so, das "Hello Prof!" ausgegeben wird, kompilieren Sie und führen Sie das Programm aus.

3. Auf der UBV finden Sie den folgenden fehlerhaften Code in `incorrectCode/incorrectCode.cpp`:

```
// This code contains two syntax errors
// fix it and get it to compile and run
// HINT: just try
// $ g++ incorrectCode.cpp -o incorrectCode
// pay attention to error messages and try to infer the error from them
#include <iostream>
int main()
{
    cout << "Hello Student!" << std::endl;
    return 0
};
```

Debuggen Sie ihn, wie im Code angegeben.

4 IDE — Integrated Development Environment

Vereinigt die Schritte Code Schreiben — Compilieren und Linken — Debug (Fehlerbehebung) — Ausführen in einheitlichem Interface. Bietet viel Hilfe, wie z.B. farbiges Hervorheben der verschiedenen Codeelemente, Hinweise auf falsche Syntax. Integriert für größerer Projekte auch die Verwaltung und Versionskontrolle (mittels `git`). Integriert teilweise AI Tools.

Wir verwenden hier “Visual Studio Code” — Open Source Version des proprietären Microsoft Visual Studio, Symbol: .

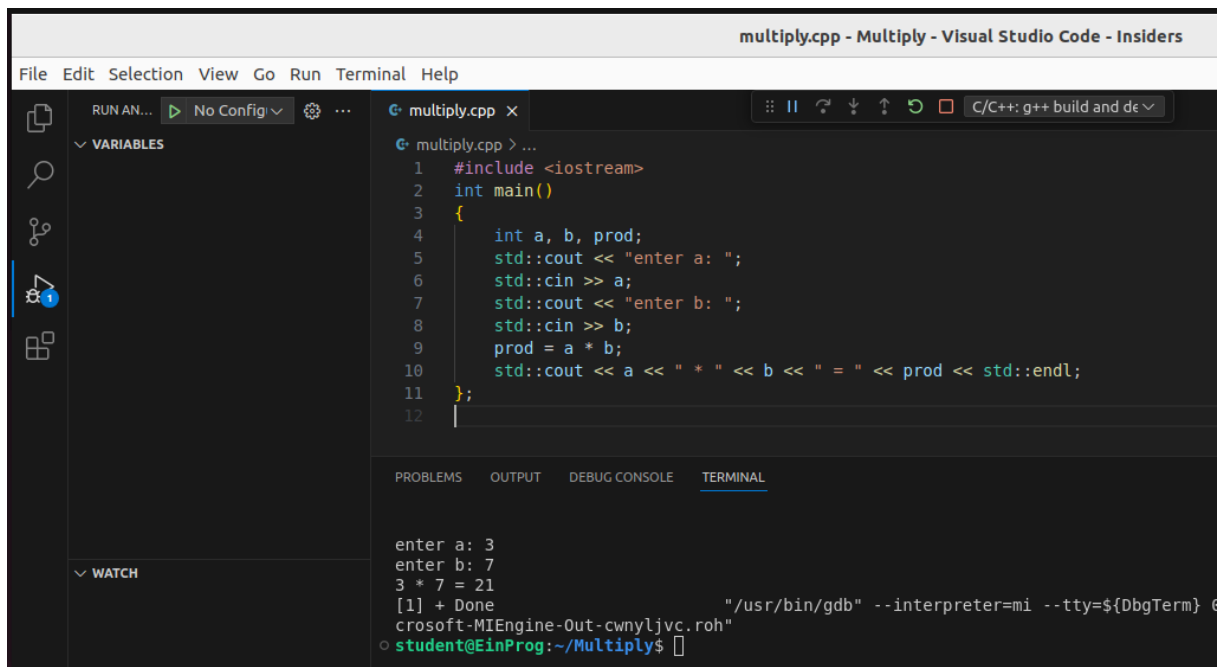


Figure 4: IDE am Beispiel multiply.cpp

5 Operatoren

Table 4: Arithmetische Operatoren

+	Addition	$x + y$	
-	Subtraction	$x - y$	
*	Multiplication	$x * y$	
/	Division	x / y	
%	Modulo	$x \% y$	17%4 gibt 1
++	Increment	$++x$ oder $x++$	erhöht Wert um 1
--	Increment	$--x$ oder $x--$	erniedrigt Wert um 1

Table 5: Zuweisungsoperatoren

	Verwendung	Entspricht
=	x=5	x=5
+=	x+=3	x=x+3
=	x=3	x=x*3
/=	x/=3	x=x/3

Table 6: Vergleiche

	Verwendung	Verwendung
==	ist gleich	x==y
!=	ist ungleich	x!=y
<	strikt kleiner	x<y
>	strikt grösser	x>y
<=	kleiner gleich	x<=y
>=	größer gleich	x>=y

6 Bedingte Ausführung: if und else

```
#include <iostream>
int main()
{
    char which;
    std::cout << "enter case A or B: ";
    std::cin >> which;

    if (which == 'A')
    {
        std::cout << "This is case A";
    }
    else if (which == 'B')
    {
        std::cout << "This is case B";
    }
    else
    {
        std::cout << "illegal case \" " << which << "\", allowed are: A,B";
    }
    std::cout << " [end of output line]" << std::endl;
}
```

Allgemeine Struktur:

```
if (logical expression1)          {... (group of) commands ... }
else if (logical expression 2)    {... different (group of) commands ... }
else if (....){...}
else                               {... do this if none of the logical expression is true ...}
```

Die else-Alternativen können auch weggelassen werden.

7 for-Loops

Dienen dazu, eine Programmteil wiederholt, aber mit veränderlichen Parametern auszuführen. Er ähnelt einem mathematischen Statement “für alle, die eine bestimmte Bedingung erfüllen”.

```
#include <iostream>
int main()
{
    for (int k = 0; k < 10; k++)
    {
        int square = k * k;
        std::cout << square << std::endl;
    }
}
```

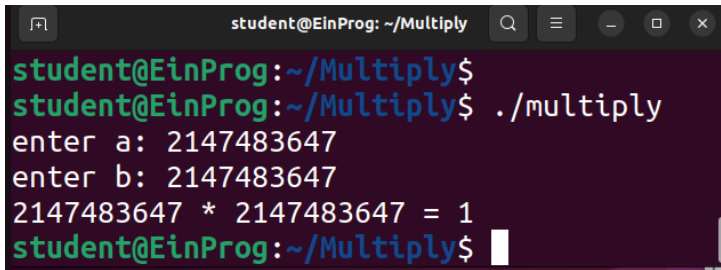
Figure 5: Loops: forLoop.cpp

Logische Struktur:

```
for(initialize variable(s); Condition to meet for execution; Update variables )
{
    ... (group of) commands to execute with current values of variables...
}
```

1. Die Initialisierung geschieht einmalig **bevor** der Loop begonnen wird;
2. Die Befehle in {...} werden ausgeführt, solange die Bedingung wahr ist
3. Der Update erfolgt **nach** der jeweiligen Ausführung von {...}.

8 Variablentypen: int, long, float, double, char, std::string



```
student@EinProg: ~/Multiply
student@EinProg:~/Multiply$ ./multiply
enter a: 2147483647
enter b: 2147483647
2147483647 * 2147483647 = 1
student@EinProg:~/Multiply$
```

Figure 6: Fehler: “overflow” von int

Weitere Typen, z.B. hier: https://www.w3schools.com/cpp/cpp_data_types.asp

Table 7: Wichtige Variablentypen und typische Speicherlänge (1 Byte = 8 Bit) und Wertebereich. **Achtung:** der C++ Standard setzt nur Minimalforderungen für die types fest, die teils niedriger sind, als hier angegeben. Die tatsächlichen Werte sind in den Compiler-spezifischen include-Files `limits.h` und `floats.h` als `INT_MAX`, `INT_MIN`, `FLT_MAX` usw. angegeben.

Name	Länge	Wertebereich
char	1 Byte	± 127 oder $[0, 2^{16} - 1]$ oder 1 ASCII Buchstabe
int	4 Bytes	$[\text{INT_MIN}, \text{INT_MAX}]$, $\pm 2147483648 = \pm(2^{31} - 1)$
long long int	8 Bytes	$[\text{LLONG_MIN}, \text{LLONG_MAX}]$, $\pm(2^{63} - 1) = \pm 9223372036854775808$
float	4 Bytes	$\pm \text{FLT_MAX}$, $\pm 3.402823 \times 10^{37}$ (ca. 6 Ziffern nach der 1ten)
double	8 Bytes	$\pm \text{DBL_MAX}$, $\pm 1.797693134862 \times 10^{308}$ (ca.. 13 Ziffern nach der 1ten)
std::string	variabel	Text und andere Zeichenketten.

8.1 Ganze Zahlen: int und long

Hier wird die binäre Darstellung von Zahlen verwendet. Von 32 Bit wird eines für das Vorzeichen benötigt, damit erhält man den Wertebereich $\pm(2^{31} - 1) = \pm 2147483648$.

8.2 Einzelne Bytes = 8 Bit: char

Mit 8 Bit kann man $2^8 = 256$ Zeichen indizieren. Die Interpretation der 8 Bit kann vom Kontext abhängig gemacht werden. Die 3 häufigsten Interpretation sind

- Buchstaben und Sonderzeichen. Insbesondere alle ASCII Zeichen, die Gross- und Kleinbuchstaben ohne Umlaute und Zahlensymbole enthalten. Dafür sind 7 Bit nötig). ASCII = “American Standard Code for Information Interchange”.
- Logische Muster mit 0 - falsch, 1 - wahr.
- Sehr kleine ganze Zahlen (kaum noch in Verwendung)

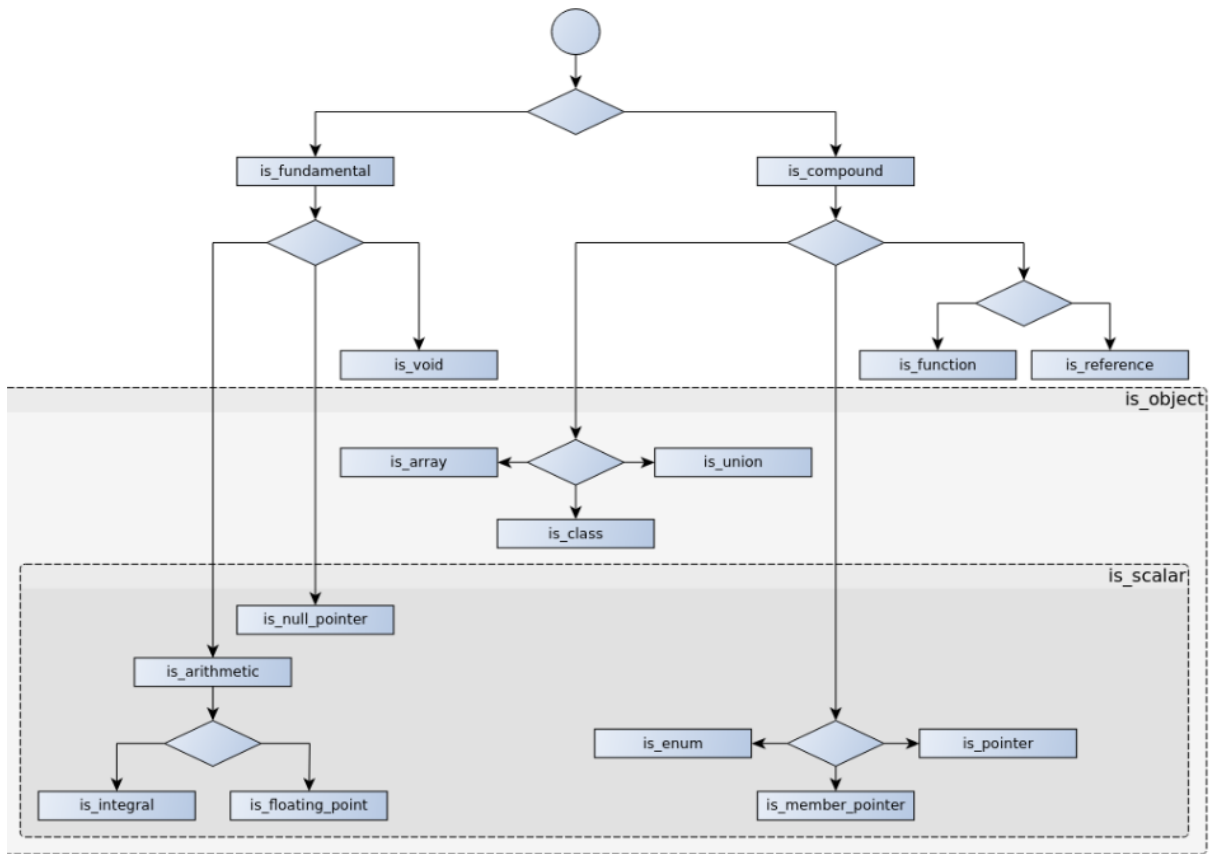


Figure 7: (**)General type structure

8.3 Zeichenketten: `std::string` aus der Standard Library

Dies ist kein “fundamental type”, der im C++ Standard festgelegt wurde, sondern ein in der C++ Standard Library enthaltener Typ, der die (halbwegs) bequeme Manipulation von Text als eine Sequenz von `char`’s erlaubt. Eine String Variable enthält eine veränderbare Anzahl von `char`:

```
std::string SomeString; // enthaelt 0 char
std::cout<<SomeString.length(); // output waere 0
SomeString="Sag doch was!"
std::cout<<SomeString.length(); // output waere 12
SomeString=SomeString+" But what?"; // wird weiter verlaengert
```

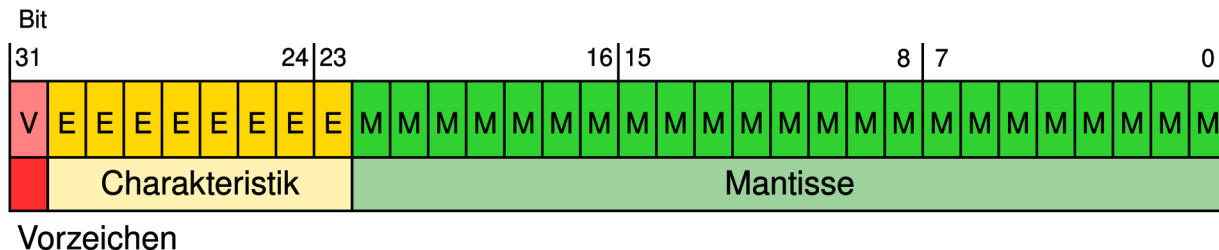
(Anmerkung: Textmanipulation ist etwas mühsam in C++, das kann, z.B., Python besser.)

8.4 Floating point (=Fließkomma) Variable: `float` und `double`

Eine Dezimalzahl mit endlich vielen Stellen kann als Paar zweier ganzer Zahlen (Exponent, Mantissee) dargestellt werden:

$$12.345 = \underbrace{12345}_{\text{Mantissee}} \times \underbrace{10^{-3}}_{\text{Basis}} \quad (1)$$

Dafür existieren Industriestandards, z.B. IEEE 754 wo von 32 Bit 8 für den Exponenten und 23 Bit für Mantissee und 1 Bit für das Vorzeichen verwendet werden:



Damit ist die Maximalzahl der Dezimalstellen auf ca. 7 beschränkt und dezimale Exponenten bis ± 38 können dargestellt werden. Die 38 ergibt sich, weil nicht der dezimale, sondern der binäre Exponent verwendet wird $2^{\pm 127} = 10^{\pm 127 \log 2} \approx 10^{38.2}$.

Der IEEE 754 für 64 Bit floating point Zahlen nutzt 11 Bit für den Exponenten, womit sich ca. 15 bis 16 Dezimalstellen und exponenten bis ca. 308 darstellen lassen.

Dies sind auch die Größenordnungen die sie für `float` (32 Bit) und `double` (64 Bit) auf üblicher Hardware erwarten können.

8.5 Truncation errors

Obwohl 7 oder gar 15 Stellen Genauigkeit für menschliches Rechnen sehr genau scheinen, gehören die Fehler durch unvollständige Darstellung von Dezimalzahlen zu einer der grösseren Probleme beim Rechnen in der Physik.

9 Header Files

Nur wenige der möglichen Funktionen und Definitionen in C++ sind von vorne herein im C++ Standard definiert und damit ohne weitere Information verwendbar. Zu den im Standard definierten Typen gehören z.B. die Variablentypen `int`, `double`, `char` und einige mehr, aber schon der Typ `std::string` wird ohne seine zusätzlichen "Header" `#include <string>` nicht erkannt. Das gleiche gilt für `std::cout`— und `std::cin`, die im Headerfile `<iostreams>` definiert sind. Beides sind Beispiele für Headerfiles der "C++ standard library", worauf auch das Prefix `std::` verweist. Wir werden andere Bibliotheken, z.B. `Eigen` mit dem Prefix `Eigen::` für lineare Algebra oder `fftw` für Fouriertransformation kennen lernen. Es ist beinahe die Essenz von C++, dass man zusätzliche Variablentypen, sogenannte Klassen `class` definiert, deren Eigenschaften und anwendbare Funktionen finden sich dann auch in einem selbstgeschriebenen Headerfile.

Die Logik hinter den Header Files ist zunächst, die Aspekte "Was?" im Headerfile vom "Wie?" im `.cpp`-File zu trennen. Nehme wir als Beispiel den Variablentyp `int`.

Was? Wenn wir ein Programm schreiben, das `int` Variable verwendet, dann müssen wir wissen, welche Operationen für diese Variablen erlaubt sind, z.B. die algebraischen Operationen `*+/-`, oder Zuweisung `int a=b` oder Output `std::cout<<a`. Für `std::string` `satz="Die Nadel liegt im Heuhaufen"` gib es zahlreiche Funktionen, z.B. `satz.lenght()` oder `satz.find("Nadel")`, die alle im Header `<string>` definiert sind. Wie die einzelnen Aufgaben umgesetzt werden, ist beim Programmieren zunächst nicht relevant.

Wie? Dies sind die tatsächlichen Algorithmen, um z.B. den Text "Nadel" in unserem `satz` zu finden. Diese werden im `.cpp`-File codiert.

Die Trennung hat mehrere Vorteile, hier einige davon:

1. Logische Klarheit: man kann im Headerfile sehen, welche Eigenschaften und Funktionen ein Objekt hat. Dies ähnelt der Definition von mathematischen Begriffen oder auch der Definition von Alltagsbegriffen.
2. Es werden nur Datentypen und Funktionen kompiliert, die tatsächlich verwendet werden. Dies reduziert insbesondere die Zeit für Kompilation, die bei großen Programmen lange, auch stundenlang, dauern kann.
3. Jeder Programmteil kann unabhängig von den anderen kompiliert werden, da alle wesentlichen Definitionen im Header vorhanden sind.
4. Algorithmen und allgemein die konkrete Umsetzung von Funktionen können leicht ersetzt werden, ohne andere Teile des Codes zu ändern

Als Nachteil kann man sehen, dass jede Funktion zweimal auftaucht: einmal im Headerfile nur mit ihren abstrakten Eigenschaften, einmal im `.cpp`-File, wo dann der tatsächliche Algorithmus steht. Bei dieser doppelten Schreibarbeit unterstützt Sie aber die IDE und verhindert Fehler.

10 Phasen des Programmierens und ihr Zeitaufwand

Man kann grob 4 Phasen bei der Erstellung eines Programms unterscheiden: Konzept und Algorithmen — Code schreiben — Fehlersuche (debug) — Korrektheitsnachweis. Der relative Zeitaufwand verschiebt sich mit wachsender Erfahrung etwa wie folgt:

Table 8: Relativer Zeitaufwand beim Programmieren — eine fiktive Tabelle

Aufgabe	Anfänger	Erfahrener
Konzept	5%	20%
Code	20%	10%
debug	70%	40%
Korrektheit	5%	30%

Die unterschiedlichen Prozentzahlen haben auch mit den unterschiedlich schwierigen Aufgaben von Anfängern und erfahrenen Programmieren zu tun. Wichtige Botschaft aus der obigen fiktiven Tabelle:

- Das Code schreiben, das Ihnen zunächst vielleicht als Hauptaufgabe erscheint, ist immer nur ein kleiner Anteil des Aufwands
- Für jeden ist Fehlersuche die langwierigste Aufgabe beim Programmieren, obwohl Erfahrung und eine gute IDE die Zeiten reduzieren können.
- Mit zunehmender Komplexität eines Programms, wird ein gutes Konzept immer wichtiger und der Aufwand dafür steigt.
- Ein “Beweis” der Korrektheit eines Programms erfordert bei wachsender Komplexität zunehmend mehr Aufwand und muss zumeist unvollständig bleiben (Warum, glauben Sie, kriegen Sie ständig Sicherheitsupdates auf Ihren Smartphones?)

11 Hardware

- CPU
- Floating Point Unit
- Memory
- Cache
- Register
- Disc
- Bus

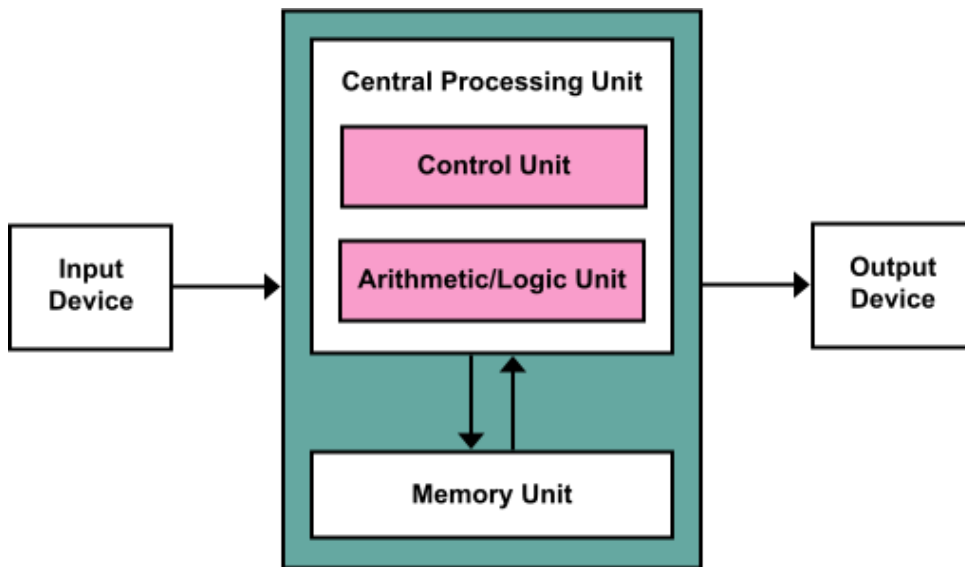


Figure 8: Hardwarestruktur (von Neumann Architektur)

12 Programmierstil: “good programming practice”

Für Lesbarkeit und — noch wichtiger — zur Reduktion von Fehlern ist ein guter und vor allem konsistenter Stil beim Programmieren wesentlich. Dazu gehören Konventionen beim Benennen der Variablen, Formatierung des Programmtexts, Grösse der Programmeinheiten, Verwendung von `##include` und bestimmter Funktionalitäten von C++, wie z.B. “forward declaration” oder `using namespace name`.

Google gibt für seine Entwickler ein umfangreiche Stilempfehlungen:

<https://google.github.io/styleguide/cppguide.html>, für grosse Projekte sicher kein schlechter Anfang. Die Regeln für diesen Kurs sind einfacher und werden je nach Bedarf ergänzt.

12.1 Variablen namen

- Verwenden Sie deskriptive Namen, z.b. `VectorComplex`, nicht `vcx`.
- Lassen Sie Visual Studio Code die Formatierung machen, alles Auswählen mittels `ctl a`, Auswahl formatieren mittels `ctl k ctl f`
- Argumente von Funktionen beginnen mit Grossbuchstaben: `void fraction(float Zaehler, float Nenner){...}`.
- Lokale Variable beginnen mit Kleinbuchstaben.
- Zusammengesetzte Namen im “camel case”: `double thisIsLocalZero=0.;`, `void myFunc(double FunctionArgument)`

12.2 Headerfiles und `#include`

13 Advanced topics (example Schrödinger equation)

These topics will be treated in the advanced version of this course by the name “Programmieren in C++”.

13.1 git — maintaining complex codes

13.2 Classes

13.3 Inheritance

13.4 Working with pointers

13.5 Templates

13.6 Containers: `std::set`, `std::map`, etc.

13.7 Parallelization: OpenMP and MPI